Ref #	Hits	Search Query	DBs	Default Operator	Plurals	Time Stamp
L1	477344	software	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L2	370232	hardware	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L3	270	DLAT	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L4	4241	TLB	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L5	1184154	target address	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L6	624476	host instruction	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L7	40182	emulat$4	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L8	477344	software	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L9	370232	hardware	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58

L10	203499	L8 and L9	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L11	270	DLAT	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L12	203499	L8 and L9	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L13	57	L12 and L11	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:58
L14	4241	TLB	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L15	57	L12 and L11	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L16	15	L15 and L14	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L17	1184154	target address	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L18	15	L15 and L14	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L19	15	L18 and L17	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59

L20	624476	host instruction	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L21	15	L18 and L17	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L22	12	L21 and L20	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L23	40182	emulat$4	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L24	12	L21 and L20	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L25	3	L24 and L23	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 14:59
L26	35	(((translation adj lookaside adj buffer) or TLB) same (consisten$4 or coheren$4) same (software or hardware) same instruction$2)	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 15:00
L27	0	22 and 26	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 15:02
L28	33	(kelly near edmund).in.	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 15:02
L29	18	(cmelik near robert).in.	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 15:03

L30	16	(wing near malcolm).in.	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 15:03
L31	41	28 or l29 or 30	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 15:03
L32	0	26 and 31	US-PGPUB; USPAT; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2004/11/09 15:03

TDB-ACC-NO: NN9106106

DISCLOSURE TITLE: Translation Lookaside Buffer Castout Queue.

PUBLICATION-DATA: IBM Technical Disclosure Bulletin, June 1991, US

VOLUME NUMBER: 34

ISSUE NUMBER: 1

PAGE NUMBER: 106 - 107

PUBLICATION-DATE: June 1, 1991 (19910601)

CROSS REFERENCE: 0018-8689-34-1-106

DISCLOSURE TEXT:

- Disclosed is a device which will decrease the performance penalty associated with heavily used congruence classes in the **translation lookaside buffer (TLB)**. It consists of a FIFO (first-in, first-out) queue into which entries from the **TLB** are placed as they age out of the **TLB**. The queue entries are functionally the same as any **TLB** entry, providing the translation benefits and adhering to **consistency** rules.

- The queue itself is larger and more complicated, entry for entry, than the **TLB**. Each queue entry holds the contents of the entry removed from the **TLB,** but in addition must keep address bits that had been used to select the congruence class for that entry. Since each queue entry must be examined in parallel with each **TLB** search, they must be implemented in the form of latches, as opposed to arrays. Compare logic looks in parallel at each entry in the queue to find logical address and segment identifier matches, outgating the corresponding absolute address field when a match is found. Additional control logic merges and manages the outputs from the **TLB** and castout queue, and blocks the **TLB** miss signal in the event of a castout queue hit. Hits in the castout queue are not restored to the DLAT, since such a policy would require additional ports to both structures.

- To maintain **consistency,** the castout queue must have purging controls functionally equivalent to those for the **TLB**. This requires a means to compare an absolute address with the absolute address component contained in each queue entry and invalidate the entries corresponding to positive compares. Since this operation is typically multi-cycle in the **TLB,** it may not be necessary for the purging logic of the queue to operate on all entries in parallel.

- Additional **hardware** is necessary to manage any additional information kept in the **TLB** and castout queue, such as GUEST IDs and control register anchors. These features are beyond the scope of this discussion.

- The magnitude of the performance benefit of this device is heavily dependent on the access pattern of the **instruction** stream being executed. Improvements are dramatic for environments which interleave accesses to the same parts of several virtual address spaces, if there are more address spaces than associativity classes in the **TLB,** but not more spaces than the sum of the number of associativity classes and entries in the castout queue. In general, this is true not only of separate spaces, but any blocks of data larger than the span of a single associativity class of the **TLB**.

TDB-ACC-NO: NN940571

DISCLOSURE TITLE: Synchronization of TLB Invalidate Broadcasts using a TLBSYNC Command

PUBLICATION-DATA: IBM Technical Disclosure Bulletin, May 1994, US

VOLUME NUMBER: 37

ISSUE NUMBER: 5

PAGE NUMBER: 71 - 72

PUBLICATION-DATE: May 1, 1994 (19940501)

CROSS REFERENCE: 0018-8689-37-5-71

DISCLOSURE TEXT:

A robust virtual memory and storage control structure can be defined by an architecture. One aspect of this specification deals with the handling of updates to the "hashed page table". The architecture describes the need for a "**TLB** invalidate" function to fully manage **coherent** updates to the table (due to the fact that **hardware** commonly "caches" the page table in a Translation Lookaside Buffer (**TLB**)).

\- In tightly coupled symmetric multiprocessing systems, this problem is complicated by the possibility of multiple processors or devices holding a copy of a page table entry in their TLBs. As a result, in order to update the page table, a mechanism for invalidating all copies must be defined. One way to accomplish this is through **software** directed interprocessor interrupts. Unfortunately, this mechanism can be quite time consuming and the **software** must be modified for different machine configurations. An alternate technique with an overall performance advantage involves automatically making the effects of the **TLB** Invalidate function occur in all processors and devices that use virtual addressing and maintain TLBs.

\- Systems that employ the use of the broadcasted **TLB** invalidate function must follow certain programming semantics, and must ensure certain control over the use of this function. In snooping bus type system implementations, if multiple devices are attempting this broadcast **TLB** invalidate function simultaneously, there exist some potential for bus deadlocks. In general, to avoid this problem, the **TLB** Invalidate function is only permitted by special **software** routines that understand such restrictions. In any case, when using the broadcast TLBI approach, it is important that a method of synchronizing the completion of the function be included.

\- One technique for solving this problem involves the use of the SYNC **instruction**. The SYNC **instruction** is a generalized synchronization operation defined in the architecture. In this context, it could be used to force a stall until all processors have completed the **TLB** Invalidate function. The problem with this solution is that the SYNC operation includes other aspects of synchronization that are unrelated to the **TLB** Invalidate. The net result of this is that the **TLB** synchronization process is subjected to additional delays associated with the SYNC operation which affects overall performance. Another drawback of this technique is that a new potential for deadlock can occur with the synchronization

operation, but since this is a generalized "SYNC", it is not reasonable to expect **software** to limit its use.

As a result, special **hardware** mechanisms must be built to avoid the deadlock potentials.

- Our solution to this problem is to create a special TLBSYNC **instruction** to be used for specifically synchronizing the **TLB** Invalidate function. In the same way the **TLB** Invalidate operation is accessible only to special **software** that understands its restrictions, the TLBSYNC is also restricted. The net result is a more effective **TLB** Invalidate synchronization function, and less complicated **hardware**.

TDB-ACC-NO: NN9204415

DISCLOSURE TITLE: Processor with Logical Instruction Cache and Physical Data Cache.

PUBLICATION-DATA: IBM Technical Disclosure Bulletin, April 1992, US

VOLUME NUMBER: 34

ISSUE NUMBER: 11

PAGE NUMBER: 415 - 417

PUBLICATION-DATE: April 1, 1992 (19920401)

CROSS REFERENCE: 0018-8689-34-11-415

DISCLOSURE TEXT:

- Disclosed is a processor which has logical **instruction** cache and physical data cache.
- A Harvard Architecture, which provides discrete **instruction** bus and data bus, is very effective, because there is no bus contention between **instruction** fetch and data manipulation. Because of the restrictions of the number of Input/Output pins, there used to be employed an Internal Harvard Architecture for microprocessors. The Internal Harvard Architecture has discrete **instruction** and data bus only inside of the chip; however, the external bus is merged to one.
- A **TLB (Translation Lookaside Buffer**) is provided for effective address conversion between the logical address and physical address.
- Fig. 1(a) shows an Internal Harvard Architecture microprocessor which has logical **instruction** cache and logical data cache. This implementation works effectively. However, when the logical address space is changed by a process switch, all caches must be purged. Moreover, it is difficult to maintain the cache **consistency** in the multiple processor configuration. These are disadvantages of the logical cache.
- Fig. 1(b) shows an Internal Harvard Architecture microprocessor which has physical **instruction** cache, physical data cache and one set of **TLB**. This implementation avoids the above problems; however, a contention occurs at the **TLB** access.
- Fig. 1(c) shows an Internal Harvard Architecture microprocessor which has physical **instruction** cache, physical data cache and two sets of **TLB**. This implementation solves the contention at the **TLB** access; however, the **hardware** cost is increased because the **hardware** cost of the **TLB** is very high.
- Fig. 2 shows an Internal Harvard Architecture microprocessor which has logical **instruction** cache, physical data cache and one **TLB**. This implementation solves the data cache **consistency** problem and the contention at the **TLB** access. It also reduces the **hardware** cost, because it does not require two sets of **TLB**. If the **instruction** is read-only under the execution, the **instruction** cache **consistency** problem does not occur.

United States. Contains confidential commercial information of IBM exempt from FOIA disclosure per 5 U.S.C. 552(b)(4) and protected under the Trade Secrets Act, 18 U.S.C. 1905.

TDB-ACC-NO: NN9405249

DISCLOSURE TITLE: Use of the SYNC Instruction to Synchronize Completion of Translation Look-aside Buffer Invalidate in a Multi-Processor System

PUBLICATION-DATA: IBM Technical Disclosure Bulletin, May 1994, US

VOLUME NUMBER: 37

ISSUE NUMBER: 5

PAGE NUMBER: 249 - 250

PUBLICATION-DATE: May 1, 1994 (19940501)

CROSS REFERENCE: 0018-8689-37-5-249

DISCLOSURE TEXT:

Disclosed is a **hardware** solution for synchronization of Translation Look-aside Buffer (**TLB**) shoot down in a Symmetric Multi-Processor System (SMP). By using the SYNC **instruction** in conjunction with the **TLB** Invalidate (TLBI) **instruction,** a method is described to ensure translation **coherency** among all processors that are contained within the SMP environment.

- The TLBI **instruction** is broadcast from the sending processor to all of the receiving processors. At this time the sending processor does not know whether each receiving processor has finished the **TLB** invalidation process since the TLBI process may take a long and varying times in each receiving processor. The sending processor is now required to broadcast a SYNC **instruction** after the TLBI **instruction,** before proceeding, in order to ensure the receiving processors have finished the TLBI process. It is up to the each receiving processors to hold off acknowledgement of the received sync operation until the received TLBI operation has completed by that processor.

Therefore, if all receiving processors have release the hold of the sync operation, the sending processor is allows to continue now knowing that the TLBI **instruction** has taken effect throughout the SMP environment.

- Translation **coherency** is maintained by the sending processor executing first the TLBI **instruction** followed by the SYNC **instruction**. The receiving processor queues the TLBI (pending execution) and meanwhile it rejects the acceptance of a SYNC until the execution of the pending TLBI is fully executed. This protocol ensures that a single processor may invalidate a **TLB** entry and the effects of the invalidation are maintained throughout any entry within the SMP complex.

TDB-ACC-NO: NN9106377

DISCLOSURE TITLE: Synonym Avoidance Cache.

PUBLICATION-DATA: IBM Technical Disclosure Bulletin, June 1991, US

VOLUME NUMBER: 34

ISSUE NUMBER: 1

PAGE NUMBER: 377 - 381

PUBLICATION-DATE: June 1, 1991 (19910601)

CROSS REFERENCE: 0018-8689-34-1-377

DISCLOSURE TEXT:

- The occurrence and cause of synonym cache lines in a digital computer cache buffer are well known in the state of the art. In general, a synonym line is located in a cache position different from the one which is derived from the requesting address. Since the synonym line stores the latest copy of the requested data, its true location in the cache must be discovered. The data access to the cache may then be completed correctly at the synonym address for a conventional cache. This discovery and re-access involves significant **hardware** expense and an additional access penalty to search the cache directory to discover the synonym, and to restart the cache with the synonym address.

- This article describes a cache structure for a general-purpose digital computer which avoids the need to provide special **hardware** to first discover and then access synonym lines with synonym addresses in the cache.

CACHE OPERATIONAL DESCRIPTION

Definition: Throughout this description, the term "Logical Address" is used. This is intended to mean any one of a variety of machine-generated addresses. It may be a real address (DAT off) or a virtual address (DAT on). It may be an **instruction** address, or an **instruction** operand address. It may be a real address due to special **hardware** references to main memory tables (e.g., segment and page tables), etc.

- The figure is a data flow of the Logical Cache with Synonym Avoidance. This data flow is composed of three subsystems:

(1) Central Processing Unit (CPU) Subsystem. This subsystem in a conventional digital processor with **instruction** units, execution units, etc., normally associated with a unit processor (UP). Multiple CPUs (not shown) interconnect via the SCE subsystem.

(2) L1 Cache Subsystem. This subsystem, which is private to each CPU, contains a set of arrays and controls, defined below, to implement a "Logical Cache". Such a cache responds directly to data requests based on "logical addresses" without need for prior address translation.

(a) L1 Cache. This is the cache data buffer and is addressed with a "Logical Address" which may be real or virtual as determined by the CPU request.

- (b) L1 Logical Directory. This directory stores an entry for each cache line in a congruence position determining the "logical address" which first requested the data from main memory. This address encodes the "principal class" for each line of data in cache.

If a line of data is valid in cache, but not locatable at a "principal class" address position, such a line is thus located in cache at one (of many) "synonym class" address position. This occurs when a subsequent cache request to a cache line is made with a different logical address than the one which first determined its positions. Note that there are special cases where a principal class line and a synonym class line have the same physical cache congruence, but different logical addresses.

The Logical Directory can provide "cache hit" data select signals to the cache only when the requested data is located at the principal class address position.

- (c) DLAT or **TLB**. This array stores virtual-to-absolute and real-to-absolute address translations for 4K byte blocks in a conventional manner. Its address congruence is determined by a requesting logical address.

- (d) Absolute Directory. This directory has one entry for each cache line. Its congruence is also determined by the logical address. The contents of the Absolute Directory are synchronized with that of the Logical Directory. Whenever a new entry is made to the Logical Directory, a new entry is made to the Absolute Directory in an identical, corresponding address position. The contents of the information stored in the two directories is different, however. The Logical Directory stores a component of the logical address (including STO for virtual addresses), whereas the Absolute Directory stores a component of the absolute address currently assigned to each virtual or real address. The Logical Directory performs address compare operations on logical addresses, whereas the Absolute Directory performs address compare operations on absolute addresses.

- (e) Cross Invalidate (XI) Logic. XI logic is used to maintain cache data **coherence**. It is invoked by the SCE for multiple CPUs sharing main memory (MP configurations), and for certain I/O operations to main memory for unit proces sors. For example, when the SCE subsystem component wishes to allocate the "exclusive" privilege of a cache line to a particular CPU, it will cause all other CPUs to mark "invalid" their copy, if any, of the same cache line.

- The XI request is signaled to CPUs by the SCE with the Absolute Address form of the data. Since the position of any cache line and corresponding directory entries is determined by a logical address form, each CPU must search all possible "synonym congruence" classes at the Absolute Directory in order to discover the principal class position of the line, if any. Upon such a discovery, the invalid bit may now be set "on" for the two corresponding entries in the Absolute and Logical Directories to complete the XI request. Setting the invalid bit "on" logically erases a line in L1 cache.

- (3) SCE Subsystem. This subsystem provides the interconnection and control logic to enable multiple CPU/L1 Cache subsystems and I/O channels to share main memory. The SCE also contains an L2 Cache and L2 Directory which are shared by the Multiple CPUs. In addition, the L1 Caches associated with each CPU operate "store-thru" to the shared L2 Cache. This means that any store operation executed at the L1 Cache is passed to the SCE and is likewise stored in the L2 Cache. For this reason, the L2 cache retains a back-up copy of any writable L1 line. Cache data management is such that any L1 "miss" causes a new L1 line fetch request to L2. Any L2 "miss" causes a new L2 line fetch request to main memory. New lines are loaded in L2 and L1 caches accordingly.

SYNONYM AVOIDANCE OPERATION

The synonym avoidance cache mechanism functions as follows.

- Referring to the figure the CPU sends logical addresses to the L1 cache subsystem. A cache "hit" at the principal class completes a normal cache access (no synonym possible). A cache "miss" at the Logical Directory indicates a principal class "miss" (a synonym line is now possible). The absolute address currently assigned to the requesting logical address is gated from the DLAT to the SCE for a new L1 line fetch request. The line will be loaded into L1 cache in a position determined by the requesting logical address, which determines its principal class for subsequent references, and new, corresponding entries will be made in the Logical and Absolute directories.
- But suppose that the requested data is already resident in the L1 cache, but at a synonym congruence position. Reloading the line in L1 cache alone would result in two copies of the same line in L1 cache, one at the principal class and another at one of the synonym classes. This is not permissible, since cache data **coherence** could not be maintained without great difficulty. Instead, as indicated in the figure, the absolute address from the DLAT used to request a new line fetch from L2, is also sent to the Cross Invalidate (XI) request block as a "pseudo" XI request. This triggers a search of the Absolute Directory at all synonym sites pertaining to the original requesting logical address.

If a synonym line exists in L1 cache, this existence will be discovered due to this "pseudo" XI request, and the corresponding entries in both the Logical and Absolute directories will be marked invalid (turn on invalid bit). This action logically erases the synonym line from cache. The final result: the line of data is relocated in L1 cache at the principal class associated with the requesting address, and the synonym line is deleted from cache.
- The major benefit of this operational scheme is the elimination of a second cache access path for synonym lines using the Absolute Directory. In prior art this path was expensive to build in **hardware** due to the need to search all the synonym classes in one (or few) machine cycles in order to minimize the cache access penalty for synonym line accesses. Instead, since the new line fetch is a multiple cycle operation, the synonym discovery search and invalidate operations may also be spread over multiple cycles. This reduces the cost and complexity of the Absolute Directory, which no longer requires many parallel address compare circuits, and eliminates the synonym address cache access path **hardware**.
- Once the line has been relocated at the principal class address, subsequent accesses to that line with the principal class address result in cache "hits". The first access "miss" which causes the line to be relocated is a multiple cycle penalty; however, this penalty may be amortized over future access requests to that line with no further synonym access penalty. Cache statistics indicate that relocating synonym lines to the principal class position is usually a good policy.

TDB-ACC-NO: NN9105301

DISCLOSURE TITLE: Clean State of MP Cache Lines for Software Control.

PUBLICATION-DATA: IBM Technical Disclosure Bulletin, May 1991, US

VOLUME NUMBER: 33

ISSUE NUMBER: 12

PAGE NUMBER: 301 - 303

PUBLICATION-DATE: May 1, 1991 (19910501)

CROSS REFERENCE: 0018-8689-33-12-301

DISCLOSURE TEXT:

- In conventional MP cache designs Valid bit (V-bit) is used to indicate whether a cache line is valid. When a cache line is modified (stored) by a remote processor, the system control (e.g., SCE) will be signaled XI-invalidate and the line will be marked invalid (e.g., with associate V-bit turned OFF). In various applications this may result in loss of concurrency. Certain techniques for **software** control of cache **coherence** (e.g., -*-) perform selected cache flushing conservatively. That is, a cache line may get flushed (invalidated) even when it is not contaminated. The benefit of such conservative flushing is the avoidance of individual XI-invalidate signaling, which is important for highly parallel MP.

In this invention we consider a more conventional environment in which the SCE (or common bus) can still carry out XI-invalidate signaling (e.g., when a line is first stored at a cache). However, a cache line need not be invalidated right away and may still be accessible until a certain point. We would like to provide the capability of flushing only those actually contaminated lines from a cache at a certain point via **software** protocols. The basic idea is to add a state to record the status of contamination at cache directories.

- Consider an MP system with processors Pi(1&i&N). Each processor has its own cache Ci . We assume that the system provides certain protocol (**instruction**) CLEANUP, which will be used to make sure that all contaminated lines are flushed out of the cache. At the cache directory, we assume an extra CLEAN state. The state may be represented in various ways. In the following we assume that the state is indicated with an explicit C-bit at each directory entry. We also assume that the processor caches are store-thru (e.g., to L2 or L3). When a processor stores into a cache line (e.g., the 1st time after the line is brought to the cache, as indicated by a Local-Change type state), the system control (e.g., SCE or common bus) makes sure that all remote caches that may contain the line get notified properly.

- When a cache line is first fetched into a cache, the associated C-bit is turned ON (meaning that it is up-to-date). When the cache control (e.g., BCE) at a processor receives a remote store signal on a line L, it checks whether L is in its cache. If so, the associated C-bit is turned OFF (meaning that the line has been contaminated by remote processors). A cache line may be validly accessed by the processor as long as it is valid (i.e., V-bit is ON), even when the

C-bit is OFF. Upon the execution of a CLEANUP, all the valid lines (with V-bit ON) in the executing processor cache are made invalid if the corresponding C-bit is OFF.

- Related Issues

(a) The CLEANUP execution may be more efficiently carried out by ANDing the corresponding V-bits and C-bits in multiplicity as bit strings.

- (b) If beneficial, it is also possible to replace the CLEAN state with its reciprocal DIRTY.

- (c) The described scheme may be operated on specific type of data lines (e.g., as in been*|, with certain tags recorded at cache directory or at **TLB,** or upon a **software** specified address ranges). Or, it may simply be operated at certain special caches (e.g., Vector Caches). Another possibility is for a program to be executed in a state (e.g., indicated in a control register) such that dirty cache lines (with C-bits OFF) are accessible. The special execution state may be turned ON and OFF with special **instructions** (by the system or user). When the state is turned OFF, the CLEANUP operations should be carried out.

- (d) Extra caution should be given to store partial merges. If a line is contaminated (i.e., with C-bit OFF), partial merge may result in erroneous data (unless the **software** can guarantee that this will not happen), in which case partial merge should be carried out at proper lower level storage hierarchy (e.g., L2). Another approach is simply to cause a refetch of (up-to-date) copy of a contaminated line when it is stored into. The 3rd approach (for keeping L1 partial merge) is to guarantee that a line can be changed by at most one processor at a time. Hence, a processor may store only into CLEAN lines in its cache. This may require authorization request (e.g, from SCE) for CH state upon the first store into a line (where CH simply means it is contaminated).

(If the line is already contaminated, a refetch should be carried out.) If the SCE detects that the requested line (for CH) is already CH at a remote cache, it should ask that remote cache give up its CH status (and turn the C-bit OFF) before granting the line fetch (with CH and CLEAN states), so that no more than one copy of the same line can be CH at the same time. A newly fetched line will have C-bit OFF if the line also stays CH at a remote cache.

- (e) For store-in cache designs the proposal approach becomes more complex. Even when the **software** may make sure that two different processors do not store into the same units (e.g., words or doublewords) of the same cache line simultaneously, the replacement of a changed line may overwrite the other processor's changes. The CH state described in (d) above may be used (except that the changed remote copy should also be transferred over).

- (f) Certain **instructions** (e.g., CS) may only be allowed to access clean lines. When a contaminated line is accessed by such **instructions,** an up-to-date copy of the line should be guaranteed (e.g., with CH state).

- (g) In the above we assume that the CLEANUP is issued by a processor to clean up its own cache. It is also possible to change it such that the execution of a CLEANUP will force the clean up of all remote caches. This may simplify programming. The CLEANUP operation may also be imbedded as the serialization requirement for certain **instructions**.

- Reference

-*- U.S. Patent 4,775,955.